# Universidad Carlos III
# Sistemas Distribuidos
# Ejercicio Evaluable 1: colas de mensajes
# Curso 2023-24

17 de marzo de 2024

Grupo: 84 y 85

Autores:
Alba Vidales Casado (100472236)
100472236@alumnos.uc3m.es - G84

Javier Sanz Diaz (100472296)
(100472296@alumnos.uc3m.es) - G85

# 1. Diseño

Nuestro diseño se basa en la creación de un servidor que recibe peticiones por colas de mensajes. El cliente manda la petición, esta petición será uno de los servicios que se han implementado en `claves.c`. Todas las funciones de la API tratan la petición en local antes de pasarla a la cola de mensajes, esto se encuentra implementado en `proxy.c`.

El servidor, que se encuentra escuchando en un bucle infinito, al recibir la petición crea un hilo al que se le pasará la información de la petición con exclusión mutua para no tener problemas de concurrencia. Este hilo llamará a la función tratarPeticion que se encargará de hacer el servicio pedido sobre una lista enlazada, que ha sido el sistema de almacenamiento que hemos elegido.

Para mejorar la solución se podría implementar un sistema de persistencia, asegurándonos así de que aunque se cierre el servidor (por desconexiones, reinicios u otros), podamos recuperar los datos que teníamos en memoria.

## 1.1. `proxy.c/claves.c`

Se encargarán de crear y rellenar los campos del mensaje de la petición, y de gestionar las colas de mensajes que se utilizarán para enviar dicho mensaje y recibir el de respuesta una vez el servicio haya realizado los servicios solicitados.

Se consideró necesaria la creación de estos archivos, ya que no debía ser el cliente como tal el encargado de gestionar colas ni mensajes.

## 1.2. `servidor.c/servicios_server.c`

Como hemos comentado, el servidor recibirá el mensaje enviado por el proxy y creará un hilo que llevará a cabo la función `tratarPeticion()`, la cuál gestionará dicho mensaje y llamará a la función lado server que corresponda. Estas funciones, encargadas de gestionar la lista enlazada se encuentran en el archivo `servicios_server.c`.

A parte de lo anterior, `server.c` realizará lo necesario para gestionar las colas, además de tratar temas de concurrencia, asegurando que en cada uno de los hilos de `tratarPeticion()` haga una copia local del mensaje que corresponda, y que las operaciones en el lado del servidor se hagan de forma atómica. Para esto tendremos dos mutex diferentes, `m_msj` que utilizará una variable de condición para controlar si se ha hecho la copia local del mensaje de petición, y por otro lado el mutex `acceso_tuplas`, el cuál asegurará la atomicidad de las operaciones haciendo lock y unlock antes y después de las mismas.

## 1.3. `cliente.c`

Únicamente, se encargará de solicitar el servicio que desee realizar. En este archivo, se han creado una serie de pruebas, las cuales van a comprobar tanto el correcto funcionamiento de las diferentes operaciones, como la concurrencia de nuestro servidor.

Para comprobar esta última, se crearon una serie de funciones que solicitaban diferentes servicios, y se llamaron en el main del cliente. Haciendo esto tendríamos varios clientes de manera concurrente.

## 2. Makefile y librería dinámica

En el Makefile se han incluido una serie reglas de depuración incluidas en la solución del segundo laboratorio que encontramos útiles en el desarrollo de nuestro servidor. Además de las reglas de depuración, está todo lo necesario para compilar la librería dinámica, el servidor y el cliente. Respecto a las opciones de limpieza se incluyen reglas para limpiar los archivos objeto, la librería dinámica y el directorio de colas de mensajes del sistema. Este último nos ha resultado muy útil sobre todo en el desarrollo del servidor, para asegurarnos de que colas que no se han cerrado correctamente no nos impidan ejecutar una nueva versión del servidor limpia. Finalmente, se incluyen reglas para ejecutar el servidor y el cliente que se detallan a continuación. Vemos a continuación cuáles sonlas anteriores:

- `all`: compila los archivos de código fuente y genera los ejecutables cliente y servidor
- `coverage`: igual que la anterior pero con opciones que permiten la generación de información de cobertura de código.
- `debug`: igual que las anteriores pero con opciones de depuración.
- `pedantic`: incluye además opciones para activar advertencias adicionales.
- `clean`: elimina los archivos generados durante la compilación y los archivos de información de cobertura de código .
- `clean-all`: misma operación que `clean` y también elimina la biblioteca compartida `libclaves.so`, si existe.
- `clean-queues`: elimina todos los archivos en el directorio /dev/mqueue/.
- `run-servidor`: compila todo y ejecuta el servidor.
- `run-cliente`: compila todo y ejecuta el cliente.

## 3. Pruebas

Para comprobar el funcionamiento del la solución implementada hemos realizado una serie de funciones de prueba. A continuación se describen brevemente cada una de ellas:

- `v_comlplete_1()`: se implementa una combinación completa de las funciones de la API.
- `v_comlplete_2()`: otra implementación de diferentes combinaciones.
- `v_set()`: se ejecutan 3 `set_value()`.
- `v_get()`: se setea un valor y se llama al get para obtenerlo.
- `v_modify()`: se modifica un valor previamente seteado.
- `v_delete()`: se elimina una clave.
- `v_exist_true()`: se comprueba que existe un valor.
- `v_exist_false()`: se comprueba que no existe un valor.
- `nv_complete1()`: una combinación de llamadas a la API en la que se falla al obtener un valor.
- `nv_length_errors()`: test en los que se pasan arrays mas largos de lo aceptado.

## 4. Consideraciones

Una de las consideraciones que hemos realizado a partir del enunciado del ejercicio es que a la hora de generar el mensaje, siempre se copiarían N valores del vector de doubles value2. De esta manera, aunque el vector tuviese más valores, se seguirían copiando N. Esto se ha hecho porque en ningún punto del enunciado se decía que esto debía producir un fallo.

# 5. Compilación y ejecución

Para compilar tanto el programa como la librería dinámica se debe ejecutar el siguiente comando:

```
make all
```

A continuación, se pueden runear un servidor y un cliente, ejecutando:

```
make run-servidor
make run-cliente
```

También se debe tener en cuenta, que están todas las funciones que se hicieron de prueba descomentadas en el main de `cliente.c`. Por tanto, alguna de estas pruebas están diseñadas para que den errores, y se deben comentar las que no interesen ejecutar.